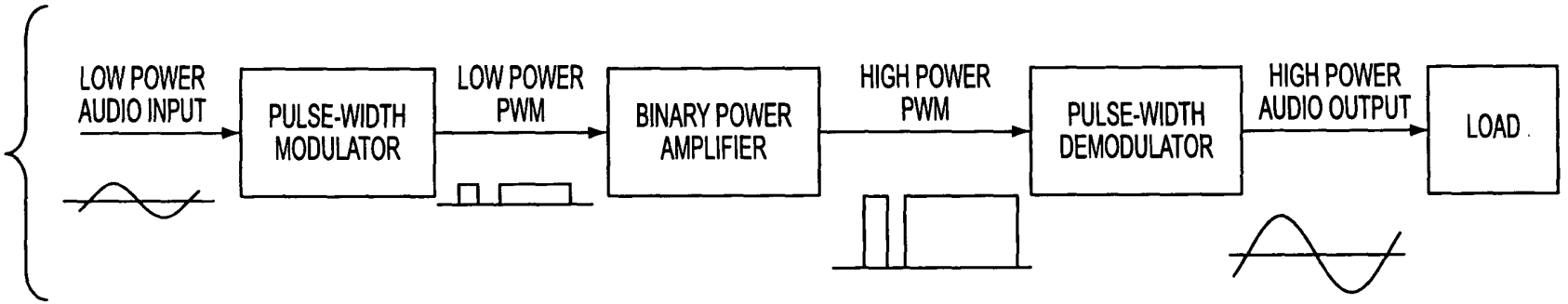

FIG. 1
PRIOR ART

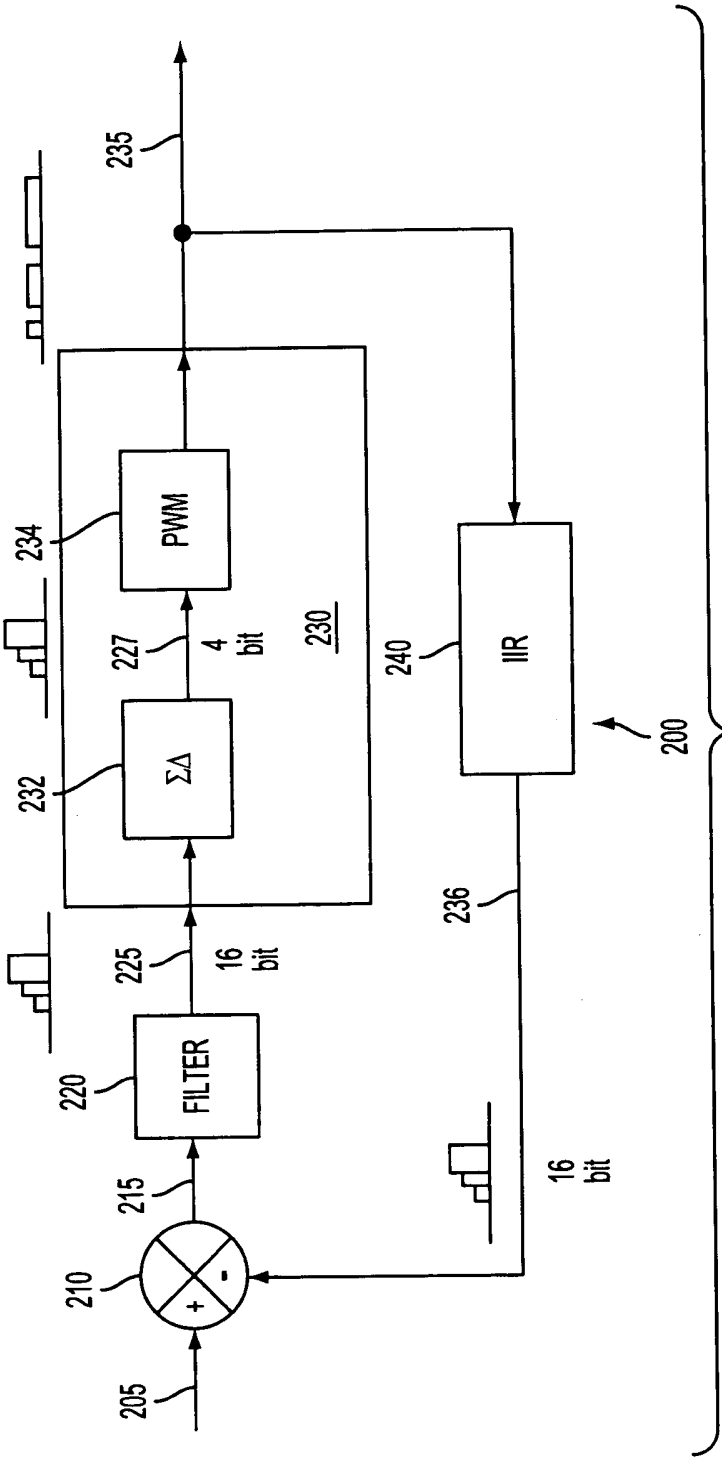

FIG. 2

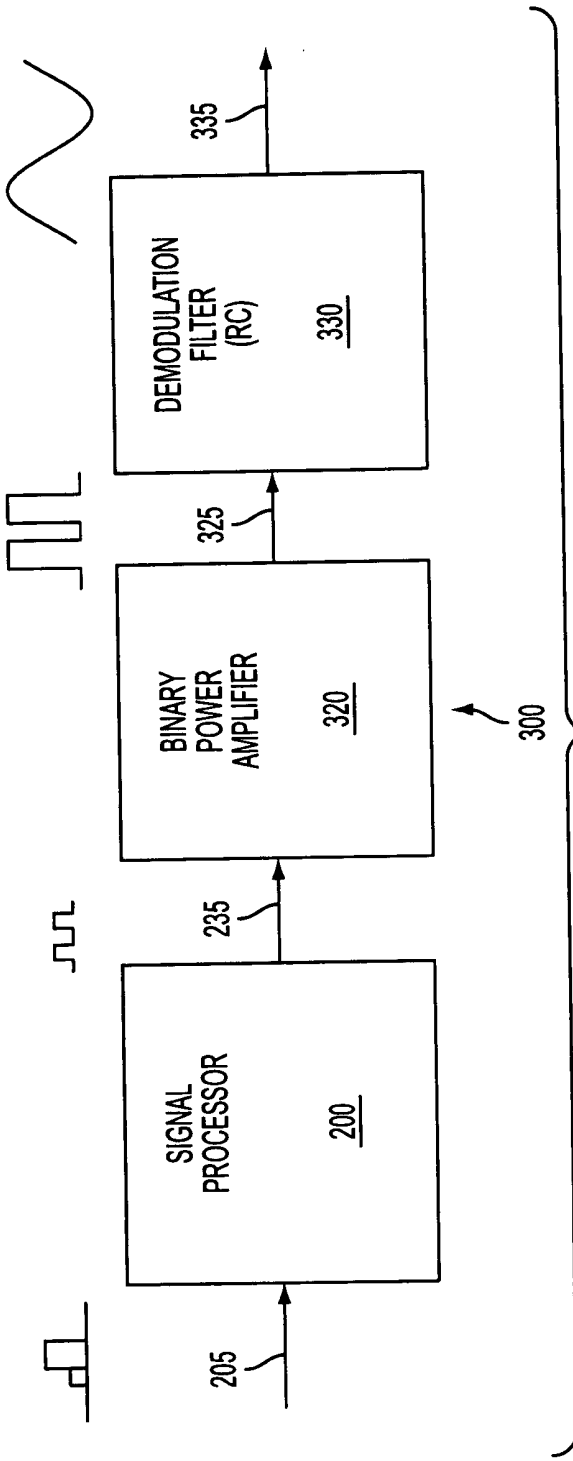

FIG. 3

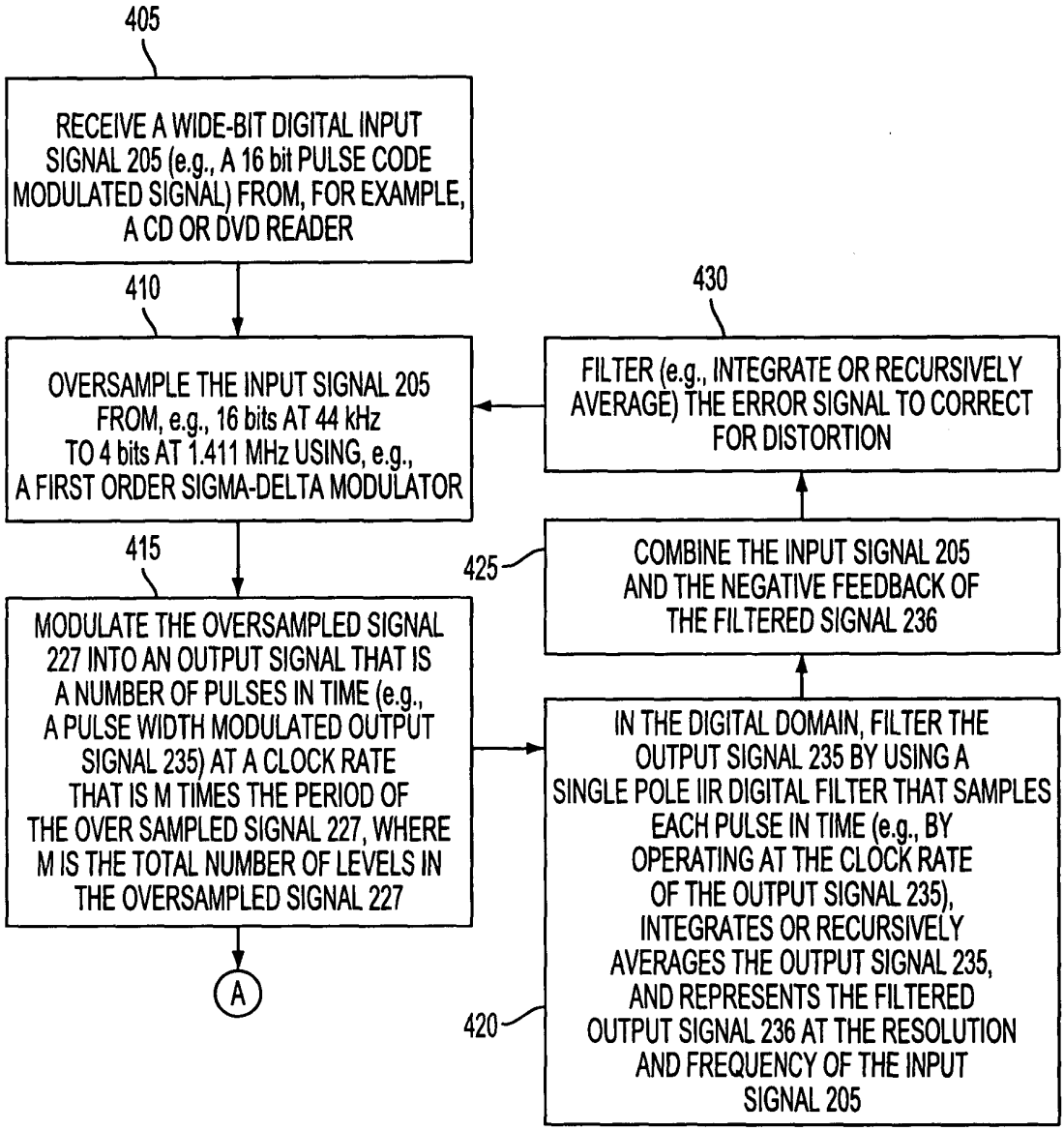

FIG. 4A

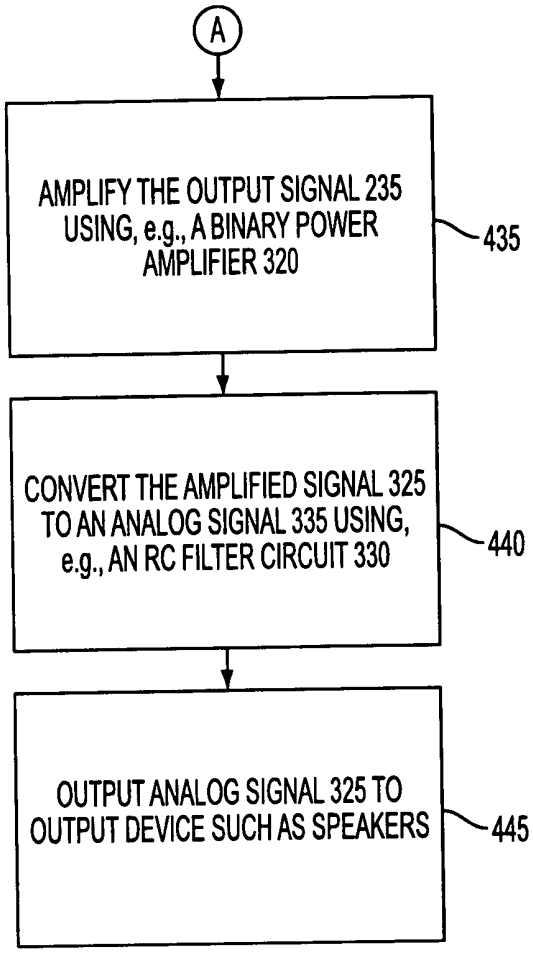

FIG. 4B

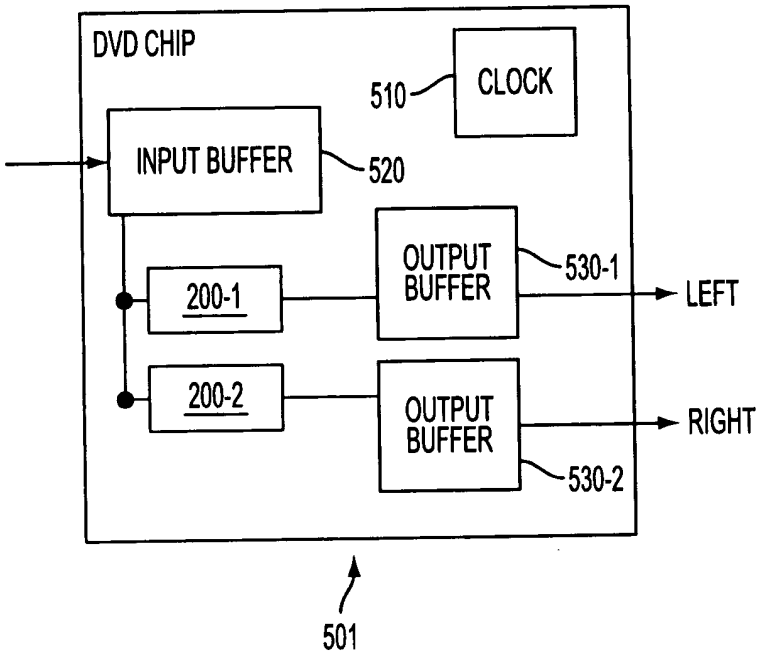

FIG. 5A

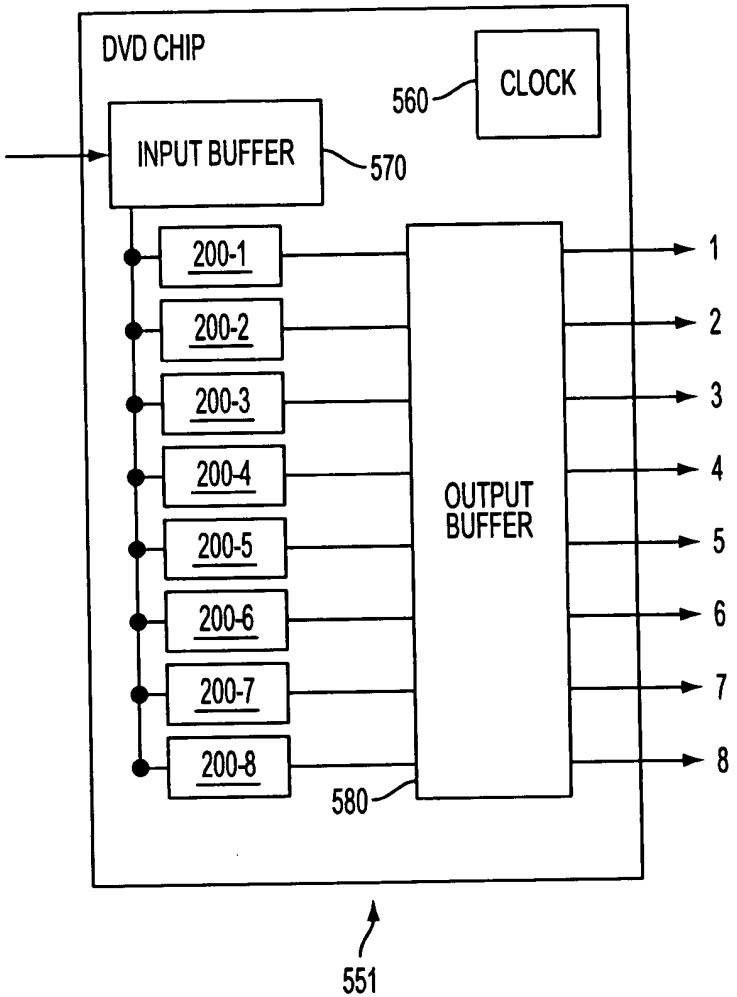

FIG. 5B

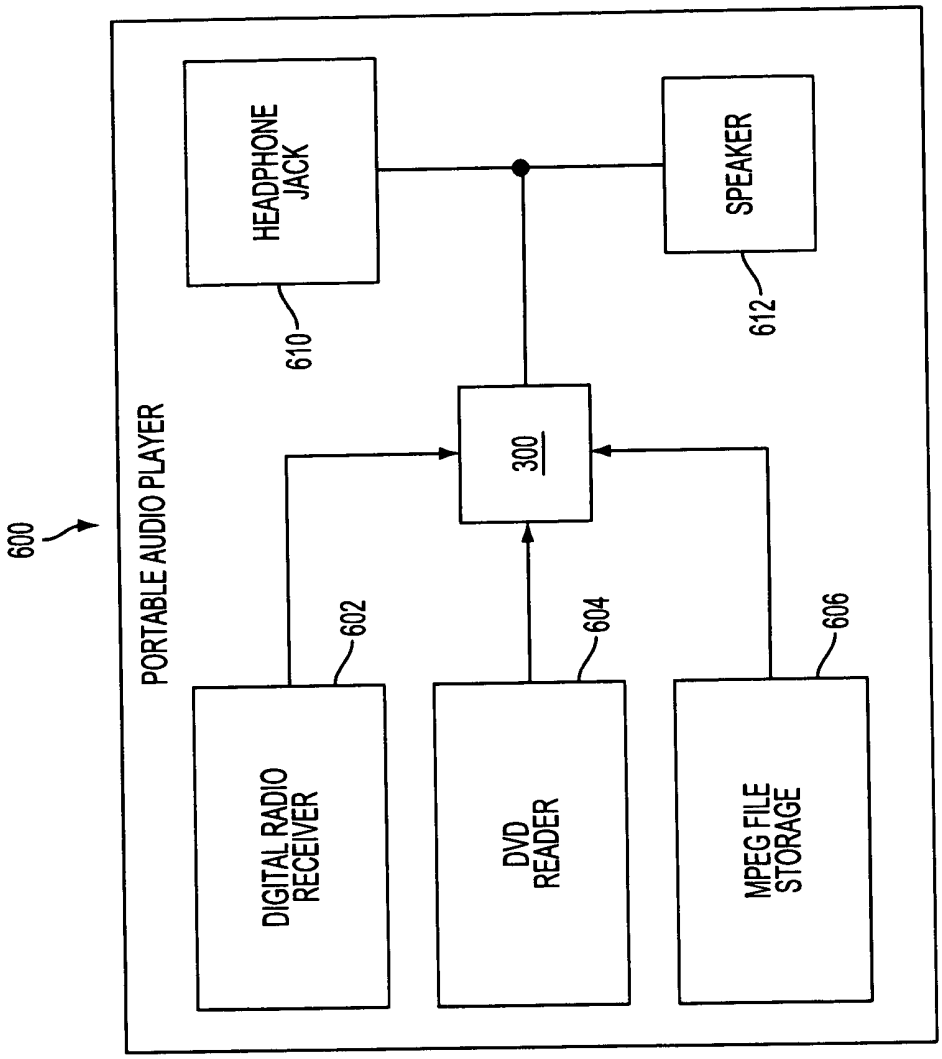

FIG. 6

```
module Example_Embodiment  (clk, clken, reset, phase, in, pwm) ;
    parameter WIDTH          = 24 ;          // Data path width
    parameter FACTOR         = 8 ;           // log2 of the divider in the ftr
    parameter PWMW           = 4 ;           // The number bits in the PWM
    parameter MI             = WIDTH - 1 ;   // Max index of bus
    parameter RST_TO         = (1 << MI) ;   // Reset state...
    input                    clk ;           // Main clock
    input                    clken ;         // A clock enable qualifier: Fclk is
                                             //   this enable expected to be about
                                             //   (16*32*Fs) = 24Mhz approx.
    input                    reset;          // Initialize asynchronous
    input  [PWMW : 0]        phase;          // PWM Phase control
    input  [MI : 0]          in;             // The input audio data
    wire   [MI : 0]          sin;            // The input scaled to -1.16db (7/8)
    output                   pwm;            // The output pwm bit
    reg                      pwm;
    wire                     pwma;           // Internal asynchronous pwmbit
    wire   [MI : 0]          fb;             // Feedback quantity..
    wire   [WIDTH : 0]       err;            // Error signal from differencer
    wire   [MI : 0]          serr;           // Scaled error value
    reg    [MI : 0]          int;            // Integrator in FB loop...
    wire   [WIDTH : 0]       sum;            // The actual sum prior to clip
    wire   [MI : 0]          nxt_int;        // Next state of clipped integrator
    wire   [MI : 0]          nxt_int1;       // Intermediate next state
    wire                force_all_one;       // Overflow and underflow bits
    wire                force_all_zero_b;
    // The forward path through the PWM to the output stream:
    pwm # (WIDTH, PWMW) pwmi (clk, clken, reset, phase, int, pwma) ;
    // The feedback path through the filter:
    ftr # (WIDTH, FACTOR) ftri (clk, clken, reset, pwma, fb) ;
    // This is one example way to clip an integrator - namely, process one
    // extra bit and look for a difference in the extra bit and the
    // sign bit like this.
    assign #3  sum                   = { serr[MI] , serr }
                                     + { int[MI] , int  };
    assign #0  force_all_one         = sum[MI] & ~sum[MI+1] ;  //  Hi => OVF
    assign #0  force_all_zero_b      = (sum[MI] | ~sum[MI+1]) ;  //  Lo => UFL
    // This next line forces the nxt_dout1 bus to all 1 if force_all_one
    // is true,  forces it all 0 if force_all_zero_b is false...
    assign #0  nxt_int1  = ( {~sum[MI] , sum[MI-1 : 0]}
                           | {WIDTH { force_all_one }})
                          & {WIDTH { force_all_zero_b }};
    // .. and then just make the signed version..
    assign #0 nxt_int = {~nxt_int1 [MI] , nxt_int1 [MI-1 : 0]} ;
    // Create the reduced amplitude input; in some cases, it may be advantageous
    // to drop the signal so that peaking at 15k is not distorted..
    assign #0 sin = in - {{3 {in[MI]}} , in [MI : 3]} ;   //   ie in - 1/8 in
    // Now create the integrator in the loop .. first assign the
    // error gain: in this case, about 3/32
    assign #1   err   = sin - fb;
    assign #1   serr  = {{4 {err[MI]}},err[MI : 4 ]}       // ie 1/16
                      + {{5 {err[MI]}},err[MI:5]};         // + 1/32
    always @ (posedge clk or posedge reset)
       if (reset)
          begin
             int   <= RST_TO;
           pwm <= 1'b0;
          end
          else if (clken)
          begin
             int   <= nxt_int;
           pwm <= pwma;
          end
endmodule // Example_Embodiment
```

FIG. 7A

```
// ========================= PWM CELL ============================
// This is an example cell that accepts a parallel input bus and creates the
// output PWM stream...
module pwm (clk, clken, reset, phase, in, out) ;
   parameter WIDTH      = 16;          // Data path width
   parameter PWMW       = 4;           // The number bits in the PWM
   parameter MI         = WIDTH - 1;   // Max index of bus
   parameter PI         = PWMW - 1;    // Max index of phase bus
   parameter PMAX       = ( (1 << PWMW) - 1) ; // Max phase count
   input                clk;           // Main clock
   input                clken;         // A clock enable qualifier: Fclk is
                                       //  this enable
   input                reset;         // Initialize asynchronous
   input  [MI:0]        in;            // The input state
   output               out;           // The output state
   input  [PWMW:0]      phase;         // This is used to allow the clk to
                                       //  run faster than the BRM so that
                                       //  the multi levels are converted
                                       //  to pulse widths. The MSB
                                       //  controls the odd/even phase, the
                                       //  other bits the PWM
   wire [PI:0]            pwm;         // Pulse width required..
   wire                 pwmi;          // . . increment by one of above
   wire [PI:0]            sum;         // Temporary - used in PWM..
   wire                 pmax;          // Phase MSB is about to change
   wire                 pdoe;          // Enable to the pdo
   // Asynchronously find when the phase MSB is about to change:
   assign               pmax = (phase[PI:0] == PMAX) ? 1 : 0;
   // Enable the pdo only when the phase MSB is about to change and
   // when the input is enabled - this generates the pdo clken signal:
   assign               pdoe = pmax & clken;
   // Modulate the input bus into the PWM Width using something
   // very similar to the pdo cell again, but note that this generates
   // an output bus of width PWM width (and note this only runs at a
   // fraction of the input clock rate)..
   pdow # (PWMW, WIDTH) pdop (clk, pdoe, reset, in, pwm, pwmi) ;
   // Now generate the pulse output by comparing the phase
   // count to the multi-bit output, however this comparison depends
   // upon the phase MSB.. Lisp code for reference:
   // (if (<= (if MSB (1+ phase) (- dith phase)) pwm) 1 -1)
   // A simple equivalent can be found by looking at the carry
   // output of the following expression:
   assign {out,sum} = ({{{PWMW{phase[PWMW]}}}} ^ phase[PI:0])
                    + pwm + pwmi;
   // Thus the output variable (out) is asynchronous - it occurs
   // shortly after the clock.
endmodule // pwm
```

FIG. 7B

```
// ========================= FILTER CELL ==========================
// This is an example filter cell for the Example_Embodiment application,
// which includes a single bit in the feedback input. It's a
// simple IIR single pole filter like this: y <= y + a(x-y) where a is
// 1/ (2^FACTOR)
module ftr (clk, clken, reset, in, out);
   parameter WIDTH      = 16;          // Data path width
   parameter FACTOR     = 9;           // The log2 of the divider
   parameter MI         = WIDTH - 1;   // Max index of bus
   input                clk;           // Main clock
   input                clken;         // A clock enable qualifier: Fclk is
                                       //  this enable
   input                reset;         // Initialize asynchronous
   input                in;            // The input state
   output [MI:0]        out;           // The output state
   reg    [MI:0]        out;
   always @ (posedge clk or posedge reset)
      if (reset)
         begin
            out <= 0 ;
         end
      else if (clken)
         begin
            out <= out
                  - {{ FACTOR { out [MI] }}, out [MI:FACTOR] }
                + {{ FACTOR+1 { ~in }}, { MI-FACTOR { in }}};
         end
endmodule // ftr
```

FIG. 7C

```
// ========================= PDOW CELL ============================
// This cell is essentially a first order ΣΔ modulator - it creates
// an output word (out) and a bit (inc) indicating the word should be
// incremented by one to minimize the noise. There is one other
// circumstance to attend to here - the 'in' may be clocked at
// a rate that differs from this clock so that the following registers the input on
// the enabled clock of this cell..
module pdow (clk, clken, reset, in, out, inc);
   parameter M  = 4;                  // The number of bits that are
                                      // "dithered"
   parameter N  = 16 ;                // The width of the input number
   parameter NI  = N - 1 ;            // Max index needed in width N
   parameter MI  = M - 1 ;            // Max index needed in width M
   parameter ME = NI - M ;            // max index of the residue: this is
                                      // the max index of the quantity
                                      // that is accumulated in the
                                      // modulo N error accumulator
   parameter RS = N - M;              // The amount to right shift the din
                                      // bus
   input            clk;              // Main clock
   input            clken;            // A clock enable qualifier: Fclk is
                                      // this enable
   input            reset;            // Initialize asynchronous
   input  [NI:0]    in;               // The input state
   reg    [NI:0]    rin;              // Registered input state
   output [MI:0]    out;              // The output state
   output           inc;              // The SD bit itself
   reg    [ME:0]    state;            // The local state of the modulo n error
   wire   [ME:0]    state_nxt;        // next state of the modulo n error
   // Just add the LSBs of the input to the running total in state,
   // allow state to overflow and keep track of the overflow bit in
   // the inc wire:
   assign  {inc, state_nxt} = {1'b0, state} + {1'b0, rin[ME:0] };
   // The output is just the msbs that are not being accumulated in
   // the state, plus 1 if the state has overflowed - as indicated in
   // the inc bit...
   assign  out = { ~rin[NI] , rin[NI-1:RS] };
   // clock like this:
   always @ (posedge clk or posedge reset)
     if (reset)
      begin
        state <= 0;
        rin   <= 0;
      end
     else if (clken)
      begin
        rin   <= in;
        state <= state_nxt;
      end
endmodule // pdow
```

FIG. 7D

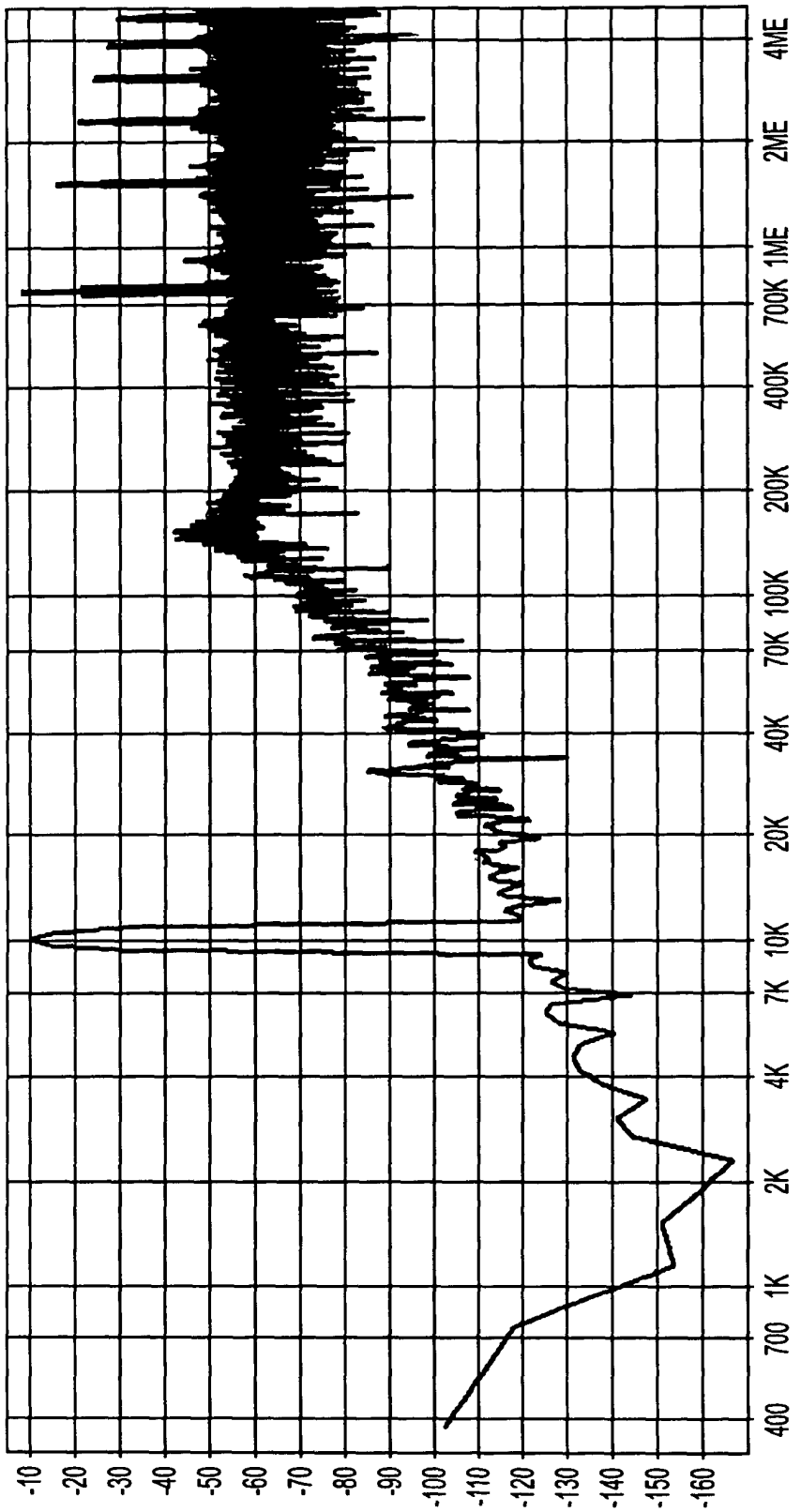

FIG. 8

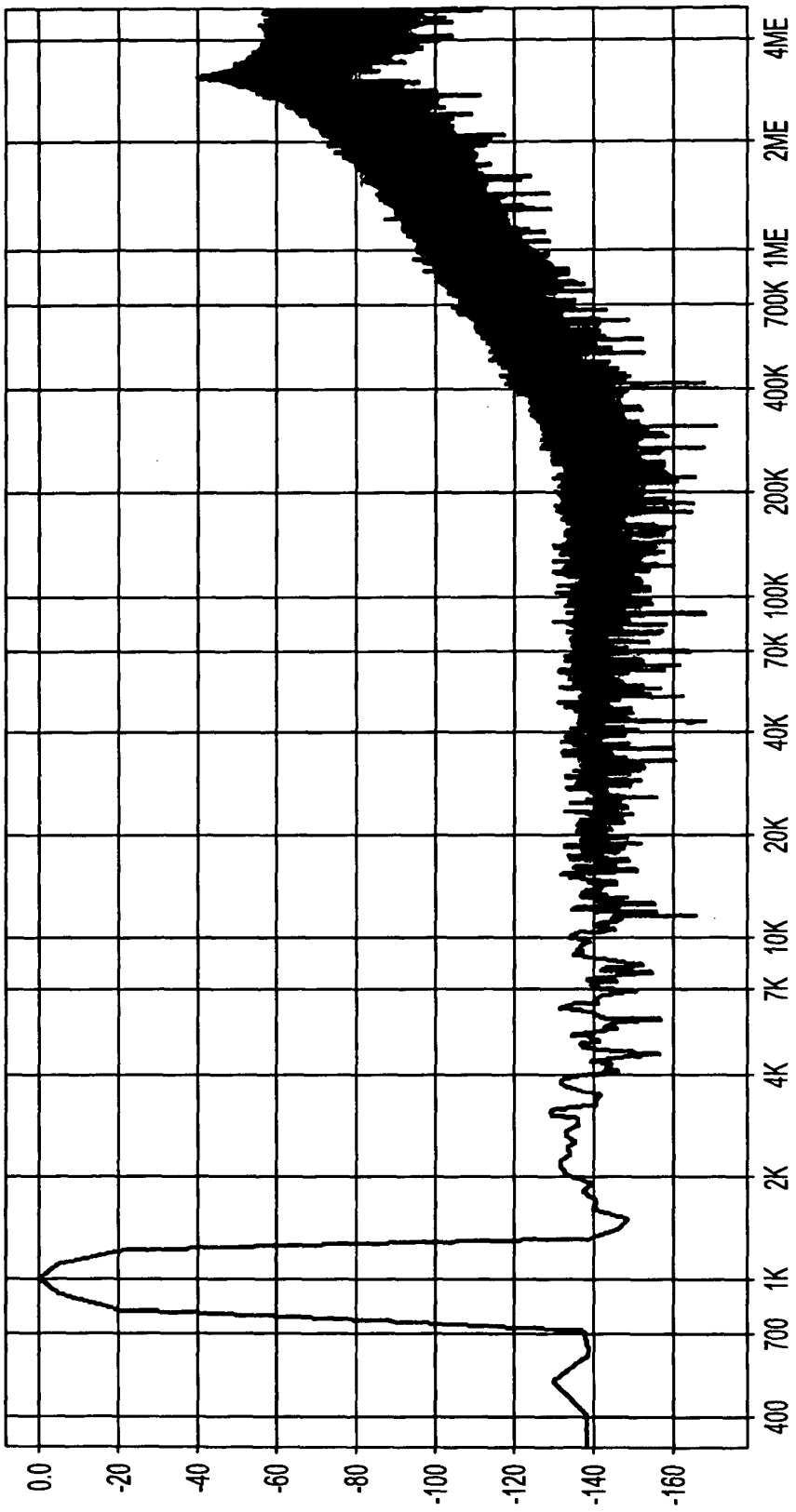

FIG. 9

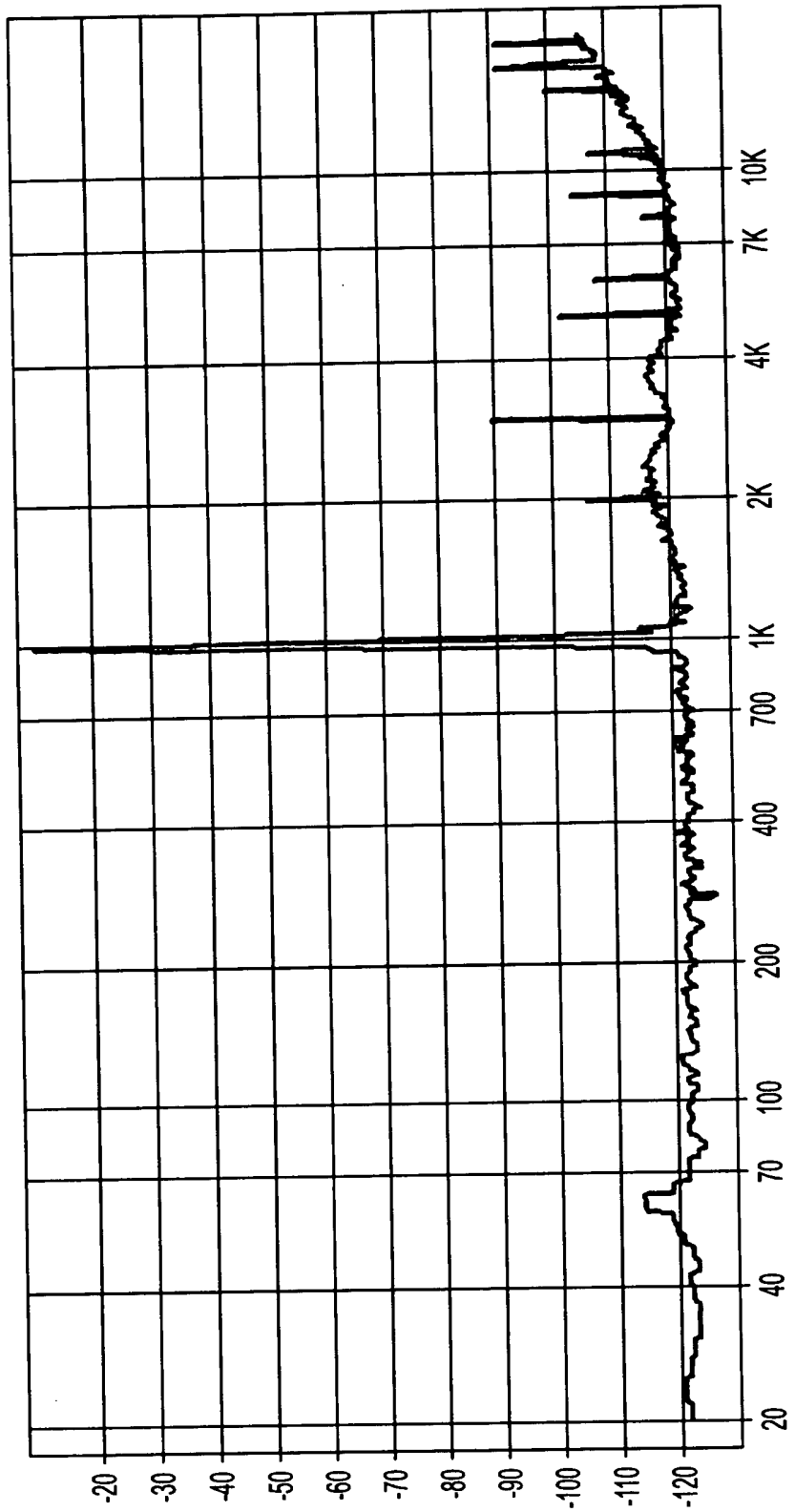

FIG. 10